CryptoMMU: Enabling Scalable and Secure Access Control of Third-Party Accelerators

Faiz Alam*
falam3@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Hyokeun Lee*
hlee48@ncsu.edu
North Carolina State University
Raleigh, NC, USA

Abhishek Bhattacharjee
abhishek@cs.yale.edu
Yale University
New Haven, CT, USA

Amro Awad
ajawad@ncsu.edu
North Carolina State University
Raleigh, NC, USA

ABSTRACT

Due to increasing energy and performance gaps between general-purpose processors and hardware accelerators (e.g., FPGA or ASIC), clear trends for leveraging accelerators arise in various fields or workloads, such as edge devices, cloud systems, and data centers. Moreover, system integrators desire higher flexibility to deploy custom accelerators based on their performance, power, and cost constraints, where such integration can be as early as (1) at the design time when third-party intellectual properties (IPs) are used, (2) at integration/upgrade time when third-party discrete chip accelerators are used, or (3) during runtime as in reconfigurable logic.

A malicious third-party accelerator can compromise the entire system by accessing other processes' data, overwriting OS data structures, etc. To eliminate these security ramifications, a unit similar to a memory management unit (MMU), namely IOMMU, is typically used to scrutinize memory accesses from I/O devices, including accelerators. Still, IOMMU incurs significant performance overhead because it resides on the critical path of each I/O memory access. In this paper, we propose a novel scheme, CryptoMMU, to delegate the translation processes to accelerators, whereas the authentication of the targeted address is elegantly performed using a cryptography-based approach. As a result, CryptoMMU facilitates the private caching of translation in each accelerator, providing better scalability. Our evaluation results show that CryptoMMU improves system throughput by an average of 2.97× and 1.13× compared to the conventional IOMMU and the state-of-the-art solution, respectively. Importantly, CryptoMMU can be implemented without any software changes.

*Both authors contributed equally to this work.

MICRO '23, October 28-November 1, 2023, Toronto, ON, Canada

ACM ISBN 979-8-4007-0329-4/23/10…$15.00
https://doi.org/10.1145/3613424.3614311

CCS CONCEPTS

• **Security and privacy → Security in hardware**; • **Hardware → Very large scale integration design**.

KEYWORDS

IOMMU, accelerator-rich architecture, access control, cryptography

ACM Reference Format:
Faiz Alam, Hyokeun Lee, Abhishek Bhattacharjee, and Amro Awad. 2023. CryptoMMU: Enabling Scalable and Secure Access Control of Third-Party Accelerators. In *56th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '23), October 28-November 1, 2023, Toronto, ON, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3613424.3614311

1 INTRODUCTION

With the increasing diversity of workloads and the significant performance and energy gap between general-purpose processors and accelerators, modern accelerator-rich architectures are getting traction in cloud systems, edge devices, and HPC systems [38, 41, 43]. While major accelerator vendors continue to develop advanced accelerators, many third-party vendors have emerged, offering cost-effective alternatives and specialized expertise [26, 36]. A variety of vendors contribute to expanding accelerator industry ecosystem by providing complementary products and services. Thus, accelerators can be manufactured with different form factors and integration strategies, such as third-party intellectual property (IP) designs embedded in a system-on-chip (SoC), soft IP designs programmed in a reconfigurable logic fabric (e.g., FPGA), and discrete accelerator chips integrated through I/O interconnects [22, 37, 61, 67, 79].

Recently, SoC designers have allowed these accelerators to directly tap into the host memory by introducing direct-memory access (DMA) attributes [9, 47, 48, 75]. Furthermore, direct access to host memory is becoming essential for modern accelerators to deal with the memory resource scarcity problem or the pointer-based programming model. Nevertheless, such DMA-enabled accelerators present significant security risks, as a compromised accelerator can potentially jeopardize the entire system [8, 66, 70]. Factors such as bugs or vulnerabilities in the device driver, an untrusted supply chain, or flaws in the accelerated kernel can substantially increase the attack surface. Therefore, it is crucial to consider these security implications when integrating third-party accelerators into a system [27, 30, 34, 45, 46, 63, 65, 80].

The Problem: To mitigate increasing security risks originating from third-party accelerators, modern systems leverage *I/O Memory Management Unit (IOMMU)* that scrutinizes accesses from third-party peripheral devices and ensures they only access their corresponding memory locations [54]. In an integrated accelerator setup, safe memory regions for accelerators are provisioned by the OS and enforced by IOMMU. IOMMU must be integrated on the trusted chip (i.e., CPU) and on the critical path of every I/O access to the host memory, including coherence messages. Thus, IOMMU is expected to handle high-throughput translation requests. However, it is expected that the pressure on IOMMU will be increased by the trend of integrating a large number of accelerators to accommodate memory-intensive custom workloads [3, 4, 50]. To address this issue, CPU manufacturers have enabled accelerators to have private TLBs for caching translations. Given that third-party vendors cannot always be trusted, implementing an efficient and secure IOMMU is critical for the advancement of accelerator-rich architectures.

The Challenge: Ensuring a robust access control mechanism is challenging because it requires verifying all traffic originating from accelerators. Different from conventional IOMMU that only handles Address Translation Service (ATS) requests on private TLB misses, a secure IOMMU also needs to check the access control (or permission) on TLB hits. Figure 1 shows that a secure IOMMU incurs 32.12× traffic overhead (i.e., ATS requests and permission check traffic normalized to ATS-only traffic). Therefore, the extra access control traffic needs to be adequately handled such that it does not saturate interconnect and memory bandwidth.

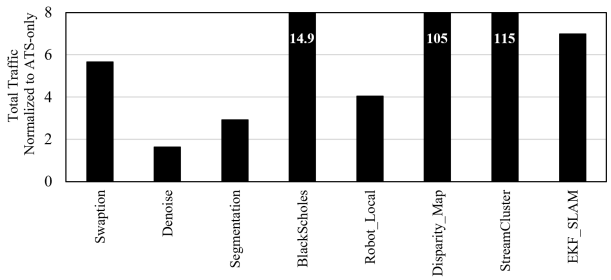

Figure 1: Total traffic (ATS+Access Control) normalized to ATS-only traffic.

To address this challenge, Olson et al., [57] have proposed Border Control (BC) that checks access permission for all traffic by decoupling page table walks and access permission. It leverages a contiguous bitmap-like structure located in the host memory to store page permissions (henceforth referred to as *protection table*) for each accelerator; hence, it does not need to walk the page table for checking access permission.

Unfortunately, BC suffers from *scalability*, *performance*, and *practicality* challenges. Border Control manages multiple protection tables in the OS, whose quantity is tied to the number of accelerators and usage contexts. Ensuring coherence between the page table and these new tables requires extending the TLB shootdown mechanism. This complicates already complex Inter-Processor Interrupts (IPI) based TLB coherence mechanism, resulting in scalability and performance issues [2, 49]. Furthermore, virtualization techniques for multi-tenant support, especially in cloud systems, amplify the complexity as they may necessitate additional synchronization between the guest and host system's page and protection tables [23]. Although Border Control introduces a physically-indexed cache, BCC, to limit main memory accesses by caching contiguous permission information, it suffers significant bandwidth overheads due to the poor locality of physical pages caused by consecutive virtual pages mapping to distant physical ones based on the system's state. This problem becomes even more severe when handling accelerators supporting multiple contexts, large memory footprints, and terabyte-sized main memory. Specifically, per-accelerator BCC configuration is not practical and scalable [31, 81]. The host should be able to support any number of accelerators; hence, putting a fixed number of BCCs in the host CPU that includes IOMMU is impractical. Consequently, *the **main challenge** in achieving scalable IOMMU implementation is to allow accelerators and devices to cache their translations yet ensure the performance of address verification at IOMMU is independent of the access pattern/locality of accelerators/devices. Also, requiring no software changes besides what is already implemented for legacy IOMMU is expected.*

Our Solution: We propose CryptoMMU to overcome the performance and scalability challenges in the prior work. It allows private address translation caching in the accelerator, thus improving scalability. CryptoMMU ensures the authenticity of pre-translated requests from accelerators through cryptographic authentication. Authentication tags generated via key-based secure hash functions provide cryptographic evidence cached with the translation in the accelerator and later verified by CryptoMMU. Consequently, CryptoMMU only needs to maintain a single authentication key per accelerator, eliminating the need for IOMMU-cached metadata, such as protection tables, dependent on access patterns. Since these keys are generated and maintained by CryptoMMU, our method does not require software changes or management of additional tables.

CryptoMMU offers two designs: a baseline version tailored for future accelerators and another for legacy accelerators. The baseline CryptoMMU employs the authentication tag, cached by accelerators alongside translations. It requires modification of the TLB structure to accommodate large authentication tags. On the other hand, for legacy accelerators whose TLB structure cannot be changed, CryptoMMU repurposes unused upper bits of the page frame number (PFN) to store a truncated tag. This allows the private TLB to cache the authentication tag with the PFN, avoiding the need for accelerator modification.

To evaluate CryptoMMU, we use gem5 PARADE simulator [10] and its High-level Synthesis (HLS)-based accelerators, as in prior work [13]. Compared to the baseline IOMMU implementation, which is similar to CryptoMMU in terms of storage overhead, CryptoMMU improves throughput by 2.97×; furthermore, it yields a speedup of 1.13× compared to the state-of-the-art scheme (i.e., Border Control).

The rest of the paper is organized as follows. In Section 2, we describe the essential background for our study. Later in Section 3, we explain why previous access control management schemes are not scalable for emerging accelerator-rich architectures. Subsequently, the proposed design, CryptoMMU, is explained in detail, followed by its evaluation results in Section 6. We discuss the overheads of

CryptoMMU in Section 7 and related works in Section 8. Finally, we conclude in Section 9.

2 BACKGROUND

2.1 Threat Model

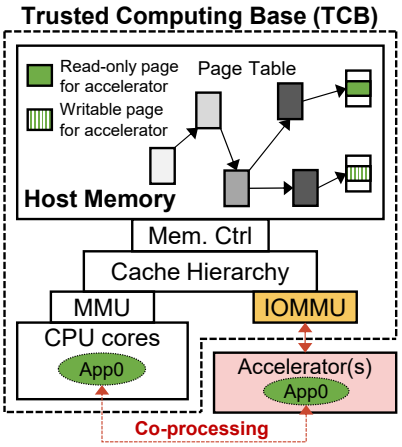

Figure 2: Our assumed threat model.

Our assumed threat model resembles previous studies of secure access control for hardware accelerators [57]. As shown in Figure 2, external accelerators outside the trusted computing base (TCB) may need to access the host memory. Untrusted accelerators, whether buggy or malicious, can potentially breach the permissions established by a trusted operating system. Such permission breaching is possible if accelerators replay outdated translations by intentionally neglecting TLB shootdown requests. Furthermore, a malicious accelerator may intercept transactions between an honest accelerator and the IOMMU, stealing address translation information to execute a man-in-the-middle attack. Consequently, bypassing memory access can result in security and reliability problems, including system crashes and the exfiltration of sensitive information. Therefore, the on-chip trusted IOMMU in TCB manages access controls. After a process running on TCB (i.e., App0 in the figure) decides the specific access permission (e.g., read-only or writable), the IOMMU will enforce the access control specified by that process. Note that we assume that host memory is inside the TCB, which can be accomplished through typical memory security protection schemes from external physical attacks [25, 78, 82] or trusted memory vendors providing point-point protection, as used in ObfusMem [7] and InvisiMem [1].

In our threat model, we consider the operating system to be trusted, which means we always assume the permissions granted by the OS are valid. Data isolation for processes executed on the accelerator or controlling the addresses accessible by the accelerator falls outside the scope of our threat model. However, the on-chip IOMMU (see Figure 2) prevents accelerators from accessing disallowed memory regions; in other words, a trusted IOMMU ensures that accelerators are allowed to read or write only the authorized pages by scrutinizing accesses coming from these untrusted accelerators.

2.2 Integration of Third-Party Accelerators

Integrating third-party accelerators in an SoC can be categorized into three approaches according to the integration moment [51]. The first approach is design-time integration, which integrates third-party designs via system buses (e.g., AXI or PCIe) before taping out the target SoC. These third-party designs may contain encrypted source files; hence, detecting bugs or hardware Trojans becomes more challenging for larger designs due to the encrypted forms of designs [58, 71]. The second approach is integration-time integration; it interfaces taped-out chips (e.g., discrete accelerator and processor chips). However, potential compromises in manufacturing chains (e.g., untrusted manufacturing/packaging/integration) may lead to bugs or hardware Trojans, thereby increasing the attack surface due to the growing reliance on these accelerators [53, 74]. The third approach is run-time integration, where SoC designers can dynamically program new accelerators on reconfigurable SoCs as users or applications themselves seek out new soft IPs that improve performance and energy efficiency [42, 68]. However, soft IP vendors potentially provide malicious or buggy designs to compromise the system security [20, 69]; moreover, soft IPs are mostly encrypted [21], further complicating security threat detection. As a consequence, we need an innovative approach that sandboxes the various types of aforementioned third-party accelerators and securely validates shared memory accesses from these accelerators.

2.3 I/O Memory Management Unit (IOMMU)

Traditionally, IOMMU provides Address Translation Services (ATS) to IO devices; it interfaces the accelerators and the main memory. Furthermore, IOMMU allows the OS to encapsulate the accelerator in its virtual memory space [50]. In such an environment, accelerators can issue requests using IO virtual addresses (*IOVA*). Subsequently, IOMMU translates IOVAs to physical addresses and checks the access controls of requested accelerators. Therefore, IOMMU also protects the system from malicious/buggy accelerators.

Similar to the MMU in processor cores, IOMMU consists of three components: a page table walker (PTW) that fetches the translations, an I/O Translation Lookaside Buffer (IOTLB) which caches translations, and page table walking caches (PTWCs) that cache intermediate levels of the page table. IOMMU is on a critical path for all memory requests issued from accelerators, and the IOTLB size is traditionally small to minimize the access latency (similar to regular TLBs [56]). Instead of using a single IOMMU, multiple parallel IOMMUs can distribute memory requests, which come at the cost of power, area, and performance. Moreover, in places where a reconfigurable logic is a part of the system, accelerator design could comprise IPs that work collaboratively. Therefore, a design to support a maximum number of accelerators is impractical.

In modern I/O interconnect protocols (e.g., PCIe), devices are distinguished using fixed hard-coded IDs linked to their ports, which the devices cannot provide for security reasons. IOMMUs allow them to supplement requests with a Process Address Space ID (*PASID*), which allows multiple processes to use the same device or accelerator concurrently. Similarly, a device can host multiple processes, each identified by its PASID, while different devices are distinguished by their hard-coded IDs. IOMMU cannot ensure data

isolation *within* a device; it only guarantees isolation between processes using the same device *when accessing* host memory with page tables. The page table to facilitate address translation can be achieved either with a ***private page table*** or ***shared page table***. We follow a shared table model as done in prior works [11, 32, 73] but note that CryptoMMU can be applied to the private page table model without modifications.

2.4 Message Authentication Code (MAC)

MAC is commonly used to verify the authenticity of data transmitted over a network, bus, or stored outside the TCB [25, 78, 82]. MAC uses a symmetric key to verify the authenticity of the data by applying a one-direction hashing function over the shared key and the message. Since the key is kept confidential, any attempt to manipulate data will fail to generate the true MAC value, which is used to authenticate such data. Once such data is fetched along with its MAC, the verifier will generate a MAC based on the data and compare it with the provided one. Thus, only data that has been authenticated and has a legitimate MAC can pass the verification check. Formally, MAC is represented as $MAC = H(Key, D)$, where Key is the authentication key and D is the data. This formula implies that any tampering by an attacker will be detected. For example, if D was tampered with, i.e., changed to D', the resulting MAC input by D' will not match upon verification, $MAC \neq H(Key, D')$, and thus the check fails. Therefore, the test will always fail unless the attacker regenerates a MAC' equal to $H(Key, D')$; however, the attacker cannot replay the true MAC value because the attacker does not know the key, which is secured in TCB. MAC values are generally large enough (e.g., 56 or 64 bits) to ensure negligible collision probability. Examples of MAC algorithms include Carter-Wegman MACs [77] generally used in the AES-GCM found in Intel SGX [25]. A relatively new family of lightweight block ciphers are PRINCE [33], MANTIS [28], and QARMA [44]. Particularly, QARMA is introduced as a part of ARMv8.3-A ISA extensions to support pointer authentication [64].

3 MOTIVATION

IOMMU becomes a major bottleneck in an accelerator-rich architecture that supports a large number of accelerators with high bandwidth requirements [13, 57]. The scalability cannot be increased by just enlarging the size of IOMMU since it sits on the critical path between I/O devices and system memory. The limited size maintains low latency, power, and area efficiency, limiting the potential to improve performance. Thus, modern I/O interconnect protocols (e.g., ATS) allow devices or accelerators to cache their own translations internally and hence directly provide the physical address to the IOMMU, i.e., bypass the IOMMU translation step. The authors of [13] demonstrated that privately caching the translations is beneficial in terms of performance; however, malicious accelerators may tamper with addresses or replay stale translations, which contradicts the purpose of using IOMMU from a security perspective.

For higher scalability, Olson et al. [57] propose *Border Control* to allow accelerators internally caching translations while ensuring security; it aims to improve the locality in IOMMU structures

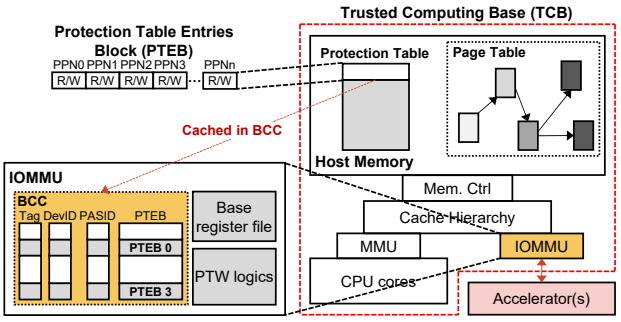

Figure 3: A high-level overview of Border Control. One protection table is allocated to each device.

through decoupling translation from access control metadata. Furthermore, access control metadata for each {accelerator, process} pair can be as little as two bits (i.e., read or write) per page, leading to the higher locality in IOMMU. As shown in Figure 3, accelerators internally cache translations and pass the cached translations to the IOMMU, whereas the IOMMU is responsible for checking if a particular physical address can be accessed.

Despite the high locality of access control metadata, two main performance challenges limit the adoption of Border Control. First, Border Control relies on IOMMU caching metadata; high contention due to many accelerators results in Border Control Cache (BCC) thrashing, which requires extra memory bandwidth. Specifically, per-accelerator BCC configuration has the issue of being impractical and non-scalable because the host should be able to support any number of accelerators. Consequently, BCC will become a bottleneck as the number of accelerators grows, highlighting the fundamental problem with Border Control in more practical accelerator-centric systems [31, 81]. Second, the flattened allocation of the protection table, which is directly indexed by physical address, may result in low spatial locality, depending on the runtime state of the system. Consequently, Border Control incurs significant performance overheads compared to unsecure ATS-only IOMMU and slows down the system performance by an average of 12.51%, as shown in Figure 4.

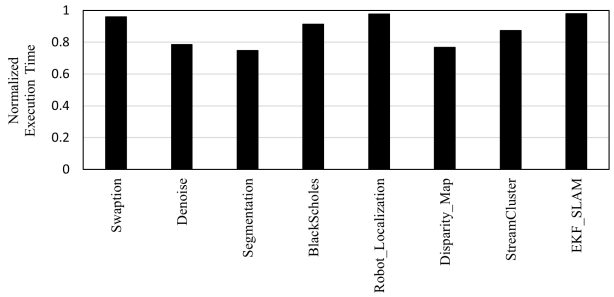

Figure 4: Performance of baseline Border Control relative to the unsecure ATS-only IOMMU.

In addition to the performance and scalability limitations of Border Control, it also introduces several other challenges. Border Control requires non-trivial changes to the OS and its memory

management sub-system to create, initialize, update, and free protection tables. To ensure consistency, the protection table must be synchronized with any updates to the permissions in the page table. In situations with high memory pressure or extensive data sharing across accelerators, the protection table base pointer may change frequently. This necessitates frequent updates to both the device table managed by kernel-level software and the protection table registers in the IOMMU. Such changes impact the software runtime for already complex TLB maintenance operations (e.g., shootdown) to additionally update the protection table. Traditionally, TLB shootdowns have been performed using software-based approaches, which involve sending Inter-Processor Interrupts to all other processors to invalidate their respective TLBs. This method can lead to performance degradation due to overheads related to invalidations, interrupts, context switches, and synchronization. To alleviate this problem, hardware-based mechanisms, such as Remote Access Requests (RAR), have been proposed. These mechanisms reduce software overhead and complexity by allowing the hardware to manage TLB shootdowns without intervention from the operating system or other software components. As a result, hardware can quickly detect changes in permissions and send targeted invalidations directly to the cores that cache the stale translations [2, 17, 49]. However, it is essential to note that implementing additional protection tables and maintaining coherence between them and shootdown mechanisms for stale protection blocks can be quite complex. Additionally, Border Control relies on the page table to populate the protection table, limiting the inclusion of page table walkers and page table walking caches within devices/accelerators [13, 60, 62].

Furthermore, Border Control incurs additional overheads. The storage overhead of large protection tables can be significant for modern systems with many multi-context accelerators using Tera-Byte regime main memory [52]. Additionally, supporting huge pages may need a complicated mechanism. A page table update of 1GB huge page may necessitate altering 1024 (assuming 4KB page size) protection table blocks. Random access within these pages can cause BCC thrashing, undermining the benefits of using large pages. In Section 7, we discussed storage overhead and support for huge pages in detail.

Border Control also faces the challenge of managing huge pages frequently used in accelerator systems. Page Table update of a 1GB huge page could necessitate permission alterations for 1,024 protection memory blocks with a 64B size. Random access within these pages can cause BCC thrashing, undermining the benefits of large pages. Efficient access control for large pages would require sophisticated mechanisms, further complicating its implementation.

Based on these observations, efficient IOMMU implementation in accelerator-rich architectures requires the following: (1) IOMMU should verify the access without the need to bring any extra metadata from host memory if a translation is cached in an accelerator; this reduces the contentions that arise due to thrashing of IOMMU's internal caches with the increasing number of accelerators. (2) No additional storage overhead such as the contiguous flat tables per {accelerator, process}, as in Border Control. (3) The IOMMU performance is independent of the access pattern of accelerators (i.e., does not rely on having a limited number of devices/processes or proximity of physical pages for neighboring virtual pages), and (4) requires no OS changes.

4 CRYPTOMMU

To obtain a secure, high-performance, and scalable IOMMU design in an accelerator-rich architecture, the following conditions must be satisfied:

(1) IOMMU will not need to bring any other per-page metadata to verify the request if a translation is cached privately by an accelerator.
(2) Minimal or zero storage overhead is incurred for storing additional metadata per page to enable access checking by the host (i.e., IOMMU).
(3) IOMMU performance is independent of the access pattern of accelerators and hence oblivious to the (spatial and/or temporal) locality of their accesses.

To achieve these criteria, CryptoMMU leverages cryptographic guarantees to enable efficient checking of the physical addresses provided by the accelerators. The philosophy of our CryptoMMU is to delegate the responsibility of proving the authenticity of pre-translated to the accelerators themselves. As a result, the IOMMU may need to fetch minimal metadata and verify it on the latency-critical path. Specifically, the request can proceed for the next operations if the accelerator can prove they are allowed access to the physical address with the provided access type; otherwise, a violation will be detected. Surprisingly, we identify *MAC* as a simple way to prove authenticity. Since most accelerators are known to be latency-tolerant, however, bandwidth-demanding, unlike CPUs, the MAC calculation latency for the address translation of accelerators is negligible. Typically, the MAC tag (or authentication tag) proves that a message is generated through a trusted party; both communication endpoints share a common session key. However, such a shared key approach is inapplicable because accelerators are not ensured to provide legitimate translations.

Unlike authenticating communication, the IOMMU acts as both a signing entity and a verification entity. The IOMMU provides translation entries that accelerators will cache upon an internal TLB miss; meanwhile, the IOMMU calculates the MAC by taking translation as one of the inputs. The calculated MAC is subsequently cached along with the translation for authentication upon hits in private TLB.

Furthermore, during the lifetime of a process, the page table might be updated, and some addresses could be unmapped. Therefore, future access to these pages must be restricted; however, accelerators can potentially conduct replay attacks by leveraging cached translation and its MAC, subsequently sending requests with stale translation. Also, relying on ATS for sending an invalidation request for such an entry to the accelerator is unsafe, as a malicious or buggy accelerator could (*intentionally*) not invalidate such a request.

Although our proposed system is expected to be popularly applied to future accelerator-rich systems, the private TLB of the accelerator requires additional fields to co-locate the MAC value along with the PFN. To prevent the modification of legacy accelerators, we also propose a novel methodology that allows CryptoMMU to overwrite the unused upper bits of the PFN; hence, the private TLB can accommodate the MAC along with the PFN without introducing new hardware resources.

As we now understand the challenges of implementing a scheme based on cryptographic authentication, we will delve into the details of CryptoMMU. First, we will describe the baseline CryptoMMU, which targets future accelerators that can adapt their internal TLBs to accommodate all the bits from the authentication tags (Section 4.1). Later, we will describe an alternative design that introduces no changes to the internal TLBs of accelerators, compatible with legacy systems (Section 4.2). Finally, we discuss how TLB maintenance operations can be handled securely in our CryptoMMU designs (Section 4.3) and further optimizations for CryptoMMU (Section 4.4).

4.1 Baseline CryptoMMU Design

Our baseline CryptoMMU relies on accelerators to provide MACs that prove the *authenticity* of the translation. The authenticity checking involves two parts: ① the physical address is allowed to be accessed by the accelerator, and ② the access type for that page is allowed by the accelerator. To prove the authenticity of such two parts, we must define the *key* and the *inputs* used to calculate the MAC per TLB entry. A straightforward definition of the key can be one authentication key for each accelerator. Such external key-based isolation may seem redundant, as isolation between processes concurrently using the accelerator is enforced; furthermore, a malicious accelerator can still leak the information internally[1]. However, if honest accelerators rely on the IOMMU to provide such checking externally, the IOMMU should support that too.

Rather than simply defining the above per-accelerator key, our authentication key is defined as a **per-{accelerator ID, PASID}** pair. The philosophy behind this definition for CryptoMMU is to detect a malicious process that is running in an accelerator and leverage a hardware bug to attempt access to the physical addresses of another process. The only case where this cannot be detected is if the hardware bug can change the PASID of the requests originating from the accelerator. Nonetheless, even in that case, CryptoMMU would still achieve the same level of security as a regular IOMMU by allowing accelerators to access permitted pages irrespective of which processes send requests (i.e., a collective set from all concurrent processes using it).

As we now know the key to be used for authentication, the other input to the authentication algorithm is the part to be authenticated. Thus, we choose the concatenation of the physical page number and access permissions in the page table entry (PTE) as the MAC generation input. Additionally, we use the virtual page number as a nonce to ensure the freshness of the generated MAC. In other words, our MAC generation will take the following form:

$$MAC_{PTE} = H(Key_{\{AccID,PASID\}}, \{PFN, R/W\}, \{nonce\})$$

The hash function H uses a per-{accelerator, process} key and takes as input the PFN extracted from the PTE, R/W permissions of the {accelerator, process} for that PFN, and virtual page number (VPN) as the nonce. Thus, as shown in Figure 5, upon a private TLB miss (Step ①) from an accelerator, IOMMU walks the corresponding page table (Step ②) to obtain the corresponding PTE. However,

[1]The accelerator is responsible for indicating which process, from those concurrently running on it, is issuing the request. Hence, a malicious accelerator can impersonate requests from another process currently running on that accelerator to access other processes. Accordingly, under our threat model, the malicious accelerator can potentially leak information between processes using it, in turn breaking their isolation.

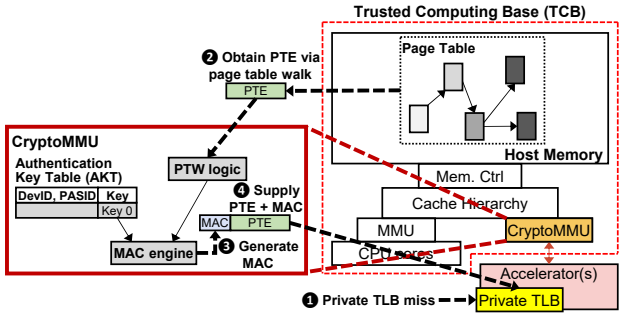

Figure 5: CryptoMMU is handling private TLB misses.

before it provides the accelerator with that PTE for future reference, it additionally augments PFN with a MAC calculated based on the attributes in the PTE resulting from the page table walk (Step ③). Similar to conventional IOMMU with ATS enabled, the accelerator will be provided with the translation information to cache internally; however, in CryptoMMU, the MAC of the translation entry is also provided (Step ④). As we can see, IOMMU acts as a signing entity for translation entries before supplying them to private TLB.

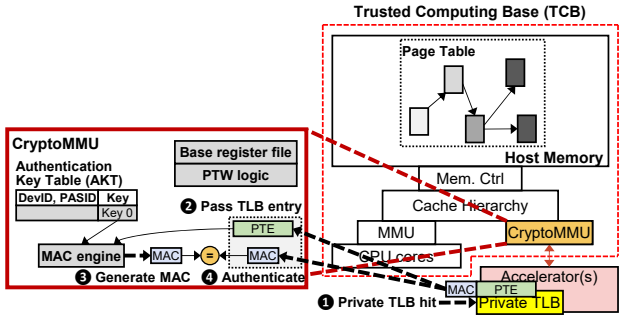

Figure 6: CryptoMMU is handling private TLB hits.

Similarly, IOMMU acts as a verification entity to verify the accesses that hit in private TLB, as shown in Figure 6. On a TLB hit (Step ①), the accelerator is prevented from directly using the pre-translated address to access the system memory without an IOMMU check. Hence, the accelerator needs to send the request containing both the physical address (along with access permissions) and the MAC for authentication to the IOMMU (Step ②). CryptoMMU obtains the appropriate key from *Authentication Key Table (AKT)* based on the Device ID (DevID) and the Process ID (PASID). The MAC engine in CryptoMMU generates a fresh MAC based on the attributes of the PTE information (i.e., physical address and permissions) provided in the accelerator's request (Step ③). The generated MAC is then compared with the MAC provided by the accelerator (Step ④). If both MACs match, it implies that the translation information provided has not been tampered with, and consequently, the system memory access is allowed (Step ⑤). If a malicious accelerator tampers with physical addresses or permissions in its private TLB or uses a stale translation, the MAC authentication will fail. Moreover, any attempt to temper MAC cached in the private TLB will also result in access failure, as discussed in Section 2.

Allocating AKT Entries: CryptoMMU requires a single authentication key per {accelerator, PASID} to ensure isolation. There are different options to realize and store such keys. In CryptoMMU, we aim to achieve the followings: (1) minimal IOMMU latency for verifying I/O requests with translated addresses and (2) no software changes. Accordingly, the CryptoMMU is responsible for creating and bookkeeping such keys. CryptoMMU uses a hardware table tagged with the Device ID and Process ID to achieve these aspects. We dub such a hardware table as *Authentication Key Table (AKT)*. The AKT leverages the unused IOTLB structure in IOMMU (since private TLB is used) and features low access latency. We reserve a certain number of entries in the IOTLB, which is typically fully associative, to be used as AKT[2], which is sufficient to allow a large number of active {accelerator, process} sessions to leverage CryptoMMU. Upon a miss in AKT, CryptoMMU checks if there exist any invalid entries in the AKT. If there exists an invalid entry, a newly generated authentication key corresponding to the {accelerator, PASID} is directly inserted into that invalid entry. On the other hand, we have two options if there is no invalid entry in the AKT: (a) avoid evicting valid entries and instead use conventional IOMMU implementation, i.e., discard the provided physical address and do the translation at CryptoMMU, or (b) use Least-Recently Used (LRU) policy to select a victim entry. While option (b) allows us to evict active sessions, it can potentially lead to frequent key re-generations for the same session if more sessions than what is cached in IOTLB are actively accessing the host memory. Although we believe it is uncommon if that is anticipated in the system, we allow CryptoMMU to be configured using option (a) or option (b) but with an additional space reserved at the bootup time in memory acting as a victim buffer for evicted sessions[3]. We assume the AKT eviction table is reserved at host memory during the bootup time to be 1MB for the whole system, allowing hundreds of thousands of active sessions without re-generating new authentication keys for a session.

MAC Calculation Granularity: In CryptoMMU, we employ a Wegman-Carter-style hash function for generating authentication tags chosen from a family of hash functions. This algorithm requires data input, nonce, and a key to produce a 56-bit output, with both input size and key being 128-bit [19, 39, 77]. To create a 128-bit block, the PTE is padded with zeros, using the virtual page number as a nonce for generating the MAC. Though some MAC algorithms do not require a nonce, our approach incorporates it to ensure the uniqueness and freshness of the MAC. Utilizing the virtual address as a nonce eliminates the need for maintaining a separate nonce with each authentication key. We anticipate that future accelerators' private TLBs will accommodate entries large enough to store the PTE (8 bytes) and its corresponding MAC (7 bytes). Subsequent sections explore possible approaches to circumvent alterations to legacy accelerators' TLB structures.

Malicious Access Violation Exception: Although the likelihood of access violations in CryptoMMU is minimal, it remains non-zero. A malicious design could potentially brute-force authentication codes and extract confidential data [76]. That is, if CryptoMMU detects a malicious access attempt by the accelerator, it will trigger an access violation exception, alerting the operating system. In response, the OS can block the accelerator and terminate all processes utilizing it, preventing any sensitive information leakage.

[2]In CryptoMMU, we use IOTLB to store AKT entries as well as invalidation buffer (will be explained in Section 4.3). The designer has the flexibility to choose the number of entries to be reserved for either of them depending on their requirements

[3]Without such a buffer changing the key of an active session renders all cached translations of that session causing verification failure and hence reverting to baseline IOMMU. In other words, it implicitly flushes the privately cached translations of the session corresponding to the evicted and re-generated entry.

4.2 CryptoMMU in Legacy Accelerators

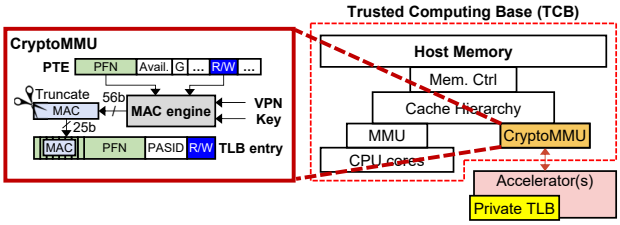

Figure 7: Operation of CryptoMMU-Legacy.

Our baseline CryptoMMU that targets future accelerator-rich systems needs to have additional fields on the private TLB of the accelerator to accommodate MAC values along with the PFN. To prevent such modifications on the legacy accelerators, we propose an alternative version of CryptoMMU, called *CryptoMMU-Legacy*. In CryptoMMU-Legacy, we allow the CryptoMMU to overwrite the unused upper bits of the PTE brought with the page table walk, as shown in Figure 7. However, the bit width of unused bits is limited; for instance, a system with 512GB of physical memory would use 27 bits of the 52-bit page frame number, leaving 25 bits of unused bits [16]. Therefore, the *truncated version of MAC* is allowed to be overwritten on unused bits in CryptoMMU-Legacy.

Security Analysis: We examined CryptoMMU's security guarantees in legacy accelerator systems by analyzing the probability of a malicious accelerator breaching its security boundaries. With a 25-bit authentication tag, the probability of malicious access is 2.98×10^{-8}, which is higher than acceptable access violation probabilities in current systems. For instance, ARM's Pointer Authentication techniques [55, 64], employed in products like the Apple iPhone XS, introduce instructions for signing and verifying virtual pointers. PAC tag sizes typically range from 3 to 31 bits, depending on the memory addressing scheme. PAC tags with 3 bits and 31 bits have access violation probabilities of 0.13 and 4.65×10^{-10}, respectively [64]. With memory tagging disabled, 11-bit PAC tags have a 4.88×10^{-4} guessing probability. Despite being developed for different threat models, these probabilities are acceptable in commercial systems. Similarly, ARM's MTE and TBI use 4 bits and 8 bits, respectively, and have higher probabilities. Therefore, we can safely adopt this probabilistic solution of CryptoMMU, which has a much higher probability of denying arbitrary memory requests.

4.3 TLB Maintenance Operation

TLB maintenance operations, namely TLB shootdown, are required upon updating the page table. Fortunately, only updating the page table requires informing the accelerator to invalidate the updated entry, because CryptoMMU avoids adding new tables (e.g., protection table). Upon an update of the page table, the OS sends an

invalidation request to all TLBs in the system to update the affected page mapping. However, since accelerators and their private TLBs are not trusted, they could intentionally attempt to replay previously valid translations by not invalidating impacted TLB entries. To prevent any replay attacks from malicious or buggy accelerators, CryptoMMU can naively generate a new key associated with {accelerator, process} upon receiving a TLB shootdown request. Subsequently, re-generating the key for MAC would prevent replay attacks by invalidating *all* the cached translations (i.e., TLB flush). Although this approach indeed allows CryptoMMU to use a lightweight MAC algorithm that does not have the additional overhead of maintaining stronger nonce (e.g., per-address counter), it would negatively impact performance since TLB flush would penalize valid cached entries.

Therefore, we propose a scalable shootdown mechanism to prevent accelerators from replaying stale translations. We introduce an on-chip *invalidation buffer* to store invalidated PTEs for each accelerator. A few entries of the IOTLB can be reserved as an invalidation buffer. On an authentication request, CryptoMMU checks the invalidation buffer to prevent the accelerator from using outdated (i.e., previously invalidated) translations; any violation results in the accelerator being blocked. When the invalidation buffer is full, CryptoMMU sends a batched TLB shootdown request to the accelerator, changes the key for {accelerator, process}, and flushes the invalidation buffer[4]. The invalidation buffer can be adopted to transparently reduce the performance penalty of individual shootdowns, effectively batching their effect on accelerators. In previous studies [2, 72], TLB shootdown occurrences due to page table updates are rare; hence, we believe that even a small invalidation buffer can significantly reduce the impact of these relatively infrequent TLB shootdown operations. Furthermore, we have evaluated the performance impact of batched invalidation in Section 6.3.

4.4 Accelerating Read Requests

A key characteristic of accelerator workloads is that it invokes thousands of address translations when transferring data between the main memory and the scratchpad memory (SPM) [32]. In such scenarios, read hits in the private TLB can be serviced by CPU caches or main memory containing a clean copy of the data block. However, write requests from the accelerator can invalidate copies in other accelerators or CPU caches, making the requester the exclusive owner of the data block. Consequently, it is crucial to thoroughly scrutinize access permissions before allowing the accelerator to update the data block to avoid data corruption/modification. In contrast, read requests may not necessitate stringent access permission evaluations. For these reasons, we propose accelerating read requests by expediting permission checks. The key idea is to overlap read access with access permission checks at CryptoMMU. This strategy enables read requests to access main memory directly while permission checks are conducted concurrently. Similarly, we optimize read misses from the private TLB by queuing up requests for an already in-flight page walk. Once a page walk request is serviced, we can generate the MAC and store it in the supplied PTE, removing redundant page walks and authentication tag generation.

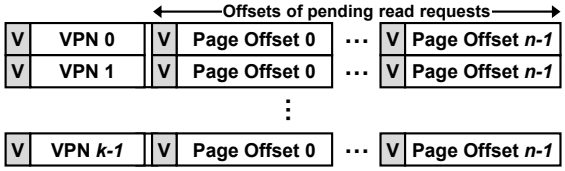

Figure 8: Data structure of RRMB.

To accelerate reads, we propose to augment CryptoMMU with a structure called the *Read Request Merging Buffer* (RRMB). As shown in Figure 8, RRMB is tagged with virtual page numbers and can hold up to k entries. The second field denotes whether the request is a read hit or a miss. Each RRMB entry allows up to n outstanding read requests to the same page and stores their page offsets. When a read request hit occurs in the private TLB, a new entry for the VPN will be allocated in the RRMB, the reads proceed to the memory hierarchy, and the data is supplied once the verification is complete. The page offsets of subsequent read-hits are stored in the RRMB, which can be serviced immediately right after the page verification. Similarly, outstanding read misses can be queued for an in-flight page walk to the same virtual page, which can be removed once the PTE is fetched and hashed. This optimization, named CryptoMMU (Read Acc.), further improves the performance of CryptoMMU, as we will show later in Section 6.1.

5 METHODOLOGY

Table 1: Simulation configuration

Processor	
CPU	1 Core, x86, OoO, 2.00GHz
L1 Cache	2-way, 32KB, 64B block, access latency: 2 cycles
L2 Cache	2MB, 8-way, 64B block, access latency: 20 cycles
DRAM Parameters	
Memory	2GB, DDR3-1600, 800MHz tRCD/tCL/tRP/tRAS: 13.75/13.75/13.75/35ns
Accelerator	
AFU	n = 4, 8, 16, 32
Private TLB	32-entry, 2-way (600B)
Latency of security scheme	
MAC Latency	20 cycles [25] (refer analysis to Section 6.2)
IOMMU	
Page size	4KB
RRMB	8 entries, 8 outstanding requests (141B)
IOTLB	64-entry, fully-associative (752B)

To evaluate the performance of our CryptoMMU design, we use gem5 PARADE, a cycle-accurate full-system simulator with high-level synthesis support [10, 15]. As shown in Table 1, we simulate an 8-issue out-of-order x86 CPU core at 2GHz with DDR3 DRAM shared by accelerators and cores. The host and the accelerator utilize a shared page table and a page table walker to manage memory access. The number of accelerators is varied between 4 to

[4]The key re-generation, i.e., forced invalidation, is necessary only when a physical page is unmapped and should no longer be accessed by the accelerator, a relatively rare event. In a unified memory programming model, for example, unmapping may occur for data consistency rather than security reasons, as accelerator access could cause coherence issues. Nonetheless, in our design, only upon a certain number of invalidations does the key need to be updated, which we expect to be extremely infrequent given its known overheads in today's systems[6, 72].

32, each of which has a 32-entry, 2-way private TLB. The PTE of the accelerator's translation is also cached in the host, as [13] does. Each accelerator contains various Function Units (FUs) designed to accelerate different algorithms for an application. A collection of *FU*s compose an *Accelerator Functional Unit (AFU)* with its own private TLB, SPM, and DMA Controller. The accelerators are tightly coupled with the host and can access the host cache as in prior works [18, 24, 29]. Application data can be tiled (split) and mapped to multiple accelerator instances, thereby leveraging data parallelism. The accelerators concurrently co-process data with CPU cores and receive virtual pointers to load/store application data. Each accelerator instance (AFU) invokes independent translation requests to load/store their data tiles from the host memory hierarchy into its software-managed scratchpad memory. However, when these accelerators issue concurrent requests, it can significantly stress the IOMMU [13, 14].

We compare our CryptoMMU against three prior approaches that support address translations to accelerators. First, the *ATS-only IOMMU*, which caches pre-translated addresses in a private TLB for subsequent accesses, and hence it is unsecure. Second, *Full IOMMU* approach, which does not allow the accelerator to cache translations, instead uses a shared IOTLB in the host. We model our Full IOMMU with a 752B 64-entry fully-associative IOTLB [56], without PTE cached in the host cache, similar to prior work [13]. Note that each IOTLB entry has 36 bits of VPN (48 bits of virtual address), 40 bits of PFN (52 bits of physical address extension), 16 bits of PASID, and 2 bits of valid and dirty information. The Full IOMMU suffers significant performance overhead from IOTLB thrashing and subsequent long latency page walks. Third, we also explored *Border Control* access control mechanism, which relies on a software-based protection table that becomes unwieldy for numerous accelerators. The size of the Border Control Cache, BCC, is the same as that of IOTLB. We allow the protection table to be cached in host caches apart from caching it into the BCC because our accelerators are tightly coupled with the host core and can directly access the host cache.

We evaluate our CryptoMMU design with two variants. First, *CryptoMMU* uses parallel MAC engines per accelerator for fast authentication tag generation and verification. Second, *CryptoMMU(Read Acc.)* accelerates read requests since they do not modify data blocks in the host. It introduces two optimizations to accelerate the read access: (1) access control verification on a read hit is overlapped with memory access to hide the access control latency; (2) subsequent read misses to an in-flight page table walk are queued and serviced when the CryptoMMU returns the MAC-augmented PTE. We leverage *Read Request Merging Buffer* (RRMB) to store both outstanding read hits and misses. Our heuristics indicate that an 8-entry RRMB containing eight pending requests is adequate for our workloads.

Workloads: We evaluate all the aforementioned approaches using accelerator benchmarks provided by PARADE [12]. These benchmarks are organized into four application domains: Medical Imaging, Computer Navigation, Computer Vision, and Commercial benchmarks, as shown in Table 2. Our benchmarks cover various ranges of memory access patterns, latency requirements, and memory footprints, hence representative of workloads of modern accelerator-rich architecture. For example, Commercial benchmarks from PARSEC are simpler and have varied access streams. Computer Navigation benchmarks have high memory footprints but regular access patterns, while Computer Vision benchmarks are latency-sensitive. Finally, Medial Imaging benchmarks have a high footprint and irregular access patterns [12].

Table 2: Description of workloads

Domain	Application	Number of FUs
Medical imaging	Denoise	2
	Segmentation	1
Commercial from PARSEC	BlackScholes	1
	SteramCluster	5
	Swaptions	4
Computer vision	Disparity Map	4
Computer navigation	EKF SLAM	2
	Robot Localization	1

6 EVALUATION

6.1 CryptoMMU Performance

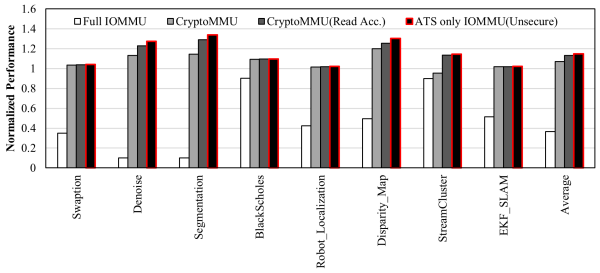

Figure 9: Normalized performance relative to the baseline Border Control.

In this section, we compare the performance of CryptoMMU, CryptoMMU (Read Acc.), Full IOMMU, and unsecure ATS-only IOMMU relative to the baseline Border Control, where the number of accelerators is set as 8 in this subsection. As shown in Figure 9, the parallel MAC generation and verification engines of *CryptoMMU* provide a low latency authentication path for accelerators with 1.07× speedup relative to Border Control. CryptoMMU(ReadAcc.), our novel read request acceleration, yields a 1.13× speedup, which is 5.6% higher than that of normal CryptoMMU. This improvement originates from overlapping access control checks with memory requests on a TLB hit and avoiding redundant page walks and authentication tag generation for in-flight requests on TLB miss. This optimization is completely safe as it is done for read requests that cannot modify/corrupt data in the main memory. As a result, CryptoMMU reduces the memory traffic of Border Control by 2.84% on average, leading to higher performance.

The unsecure ATS-only IOMMU has an improvement of 1.15×, whereas Full IOMMU suffers from a slow-down of 62% relative to Border Control. Additionally, Full IOMMU has a slow-down of 66.42% relative to CryptoMMU(Read Acc.). In particular, the unsecure ATS-only IOMMU has the highest speedup because all the accelerators have private pre-translated addresses to access the physical memory. In contrast, the Full IOMMU does all address

translations at the IOMMU, leading to considerable performance degradation. Medical imaging benchmarks, such as Denoise and Segmentation, exhibit large page reuse distances that can cause thrashing in the Border Control Cache (BCC). In contrast, CryptoMMU effectively achieves higher speedup by hiding verification latency. StreamCluster and DisparityMap involve multiple iterations using distinct accelerators on identical input data. This results in protection table entries for different accelerators constantly evicting BCC entries, leading to CryptoMMU's superior performance, as it remains unaffected by such access patterns. BlackScholes and Swaption are benchmarks with low memory footprints, primarily involving simple arithmetic operations. Consequently, they exhibit smaller speedup improvements than other, more complex benchmarks. On the other hand, EKF SLAM and Robot Localization are computer navigation benchmarks that utilize large data arrays with consistent access patterns. These patterns fit well within the BCC, resulting in a comparable performance for all schemes except for Full IOMMU.

6.2 Sensitivity Analysis

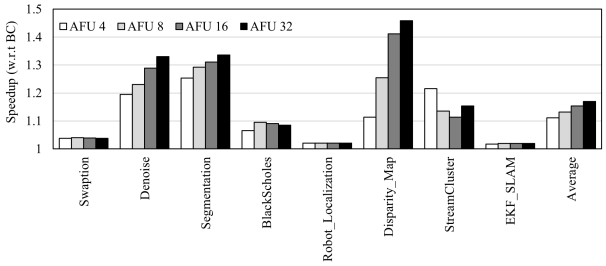

Figure 10: Sensitivity to the number AFUs.

In this subsection, we rigorously conduct various sensitivity studies regarding different architecture parameters in CryptoMMU.

Varying the Number of AFUs: We study the impact of different numbers of *AFUs* on CryptoMMU. Figure 10 shows the speedup of CryptoMMU with Read Acceleration relative to baseline Border Control. In general, growing the number of accelerators improves data parallelism, resulting in improved performance on average. CryptoMMU performance improves with the increasing number of AFUs, with average speedups of 1.11×, 1.13×, 1.14×, and 1.16× for 4, 8, 16, and 32 AFUs, respectively. The Border Control design is not well-suited for scaling to several accelerators sharing a fixed-size BCC, which may result in thrashing. As the number of accelerators increases, they compete for a fixed number of cache lines for permission checks, leading to additional protection table accesses to the main memory.

Particularly, *Denoise* and *Segmentation* show the highest speedups due to their irregular access patterns, which cause thrashing of the BCC. *Disparity Map* is a latency-sensitive computer vision application that performs poorly due to long-latency protection table access. In contrast, *Swaption* and *BlackScholes* are low-footprint benchmarks with regular access patterns which do not put much pressure on the BCC; hence, the improvements from data parallelism dominate with increasing AFU. Finally, *StreamCluster* has high data locality, as it sweeps streams of large arrays, resulting in a high BCC hit rate. The performance improvement between AFU counts of 16 and 32 suggests degradation due to BCC thrashing outweighs the benefits of increased data parallelism.

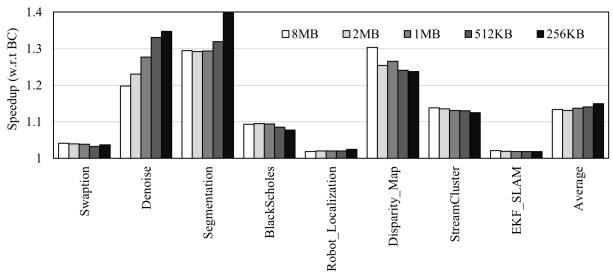

Figure 11: Sensitivity to varying sizes of LLC.

Varying LLC Size: The accelerators in our design are tightly coupled with the CPU core and share L1 cache and LLC. Therefore, we are caching the accelerator's PTEs, data, and protection table (in Border Control) in L1 and LLC. Figure 11 presents the performance improvement of CryptoMMU (Read Acc.) relative to Border Control with varying LLC sizes. Generally, the relative performance of CryptoMMU improves as the LLC size decreases. CryptoMMU yields average speedups of 1.13×, 1.12×, 1.13×, 1.14×, and 1.15× when the LLC sizes are 8MB, 2MB, 1MB, 512KB, and 256KB, respectively. As the LLC size decreases below 2MB, the relative performance of CryptoMMU improves due to reduced caching of security metadata in Border Control. However, a large 8MB LLC results in increased cache access time, degrading the performance of Border Control due to the higher latency while fetching the protection table from the LLC.

Figure 11 shows that *Denoise* and *Segmentation* exhibit the highest speedups, similar to results in Figure 10. Their irregular access patterns cause BCC thrashing, leading to frequent protection table block fetches in the host cache. These protection table blocks create cache pollution in the host by evicting useful blocks. However, an 8MB LLC alleviates cache pollution for *Segmentation*. In contrast, navigation workloads like *Robot Localization* and *EKF-SLAM* display lighter cache pollution due to more regular access patterns.

DisparityMap demonstrates better performance with 1MB LLC than 2MB. This is because L1 and LLC are inclusive caches; a smaller LLC can result in fewer back invalidations in L1 upon eviction in LLC. *BlackScholes* displays similar performance tendencies. Conversely, *StreamCluster* extremely regular access pattern allows protection table metadata to be cached longer before eviction, providing low service latency compared to CryptoMMU, which requires MAC generation and verification.

Varying MAC Latency: CryptoMMU incurs additional MAC latency for every TLB hit (i.e., MAC verification) or a miss (i.e., MAC generation). The MAC latencies are determined by various factors, including input size, key size, area budget, power budget, security margin, and latency requirements. A recent work, QARMA, incorporated in Arm's *Pointer Authentication Code* [64], consistently reported latency of less than 10ns for 64-bit and 128-bit blocks

across latency and area-optimized designs [5]. For AES-GCM's recent commercial implementation, e.g., in Intel's memory encryption engine, the throughput is 1 AES block per cycle, and for polynomial multiplier (to calculate the MAC) is one multiplication per cycle with a frequency of 3.2GHz [25]. Thus, 10ns in Table 1 is a conservative assumption for AES-GCM in commercial products. We choose the design latency for both MAC generation and MAC verification to be 10ns. To show the robustness of our design in hiding the authentication latency, we stressed our model by varying the MAC latency up to five times. Our results show that the parallel authentication tag generation/verification scheme of CryptoMMU is very robust in hiding the latency of MAC authentication (i.e., both MAC generation and verification) and only show a slow-down of 3.76% at 50ns relative to the design latency.

6.3 CrytoMMU Overheads

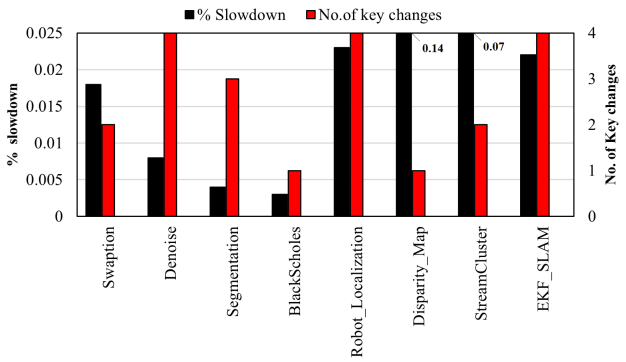

Figure 12: Percentage slowdown of CryptoMMU(Read Acc.) with batched invalidation for 200 invalidations per second.

In this section, we present a quantitative evaluation of CryptoMMU, specifically focusing on the impact of batched invalidation to prevent replay attacks. As shown in Figure 12, we evaluated the number of batch invalidations (i.e., number of key re-generations) and corresponding percentage slowdown in CryptoMMU(Read Acc.). CryptoMMU does batch invalidations of TLB entries once the invalidation buffer is full. In our design, we allocate half of the IOTLB entries, specifically 32, as an invalidation buffer for eight accelerators. Each accelerator is assigned four entries of the invalidation buffer, allowing them to store up to 8 invalidated pages. It is important to highlight that CryptoMMU provides flexibility to configure a larger invalidation buffer for performance-critical workloads. We assumed 200 invalidations per accelerator per second based on the prior work [57]. Based on our evaluation, *Denoise* receives 27 TLB shootdown requests, which implies CryptoMMU must execute four key changes. The average slowdown is 0.017% relative to the design without batched invalidation (i.e., changing the MAC key). *DisparityMap* experiences the most significant slowdown, a modest 0.14%, given its sensitivity to latency. *StreamCluster*, on the other hand, has a high locality in the device's private TLB; hence, it experiences a slowdown by 0.07% due to forced flushing of private TLB.

7 DISCUSSION

Estimation of Storage Overheads: CryptoMMU does not store security metadata in main memory but may use minimal storage for cryptographic keys. It relies on unused or extra TLB bits for authentication tags. In contrast, Border Control uses physically tagged protection tables per accelerator, causing storage overhead to scale with processes, accelerator units, and main memory size. For a system with 1TB of memory, 16 accelerators, and 20 concurrent processes, Border Control needs 10GB for protection tables, while CryptoMMU needs just 5KB for 320 128-bit keys per accelerator-process pair.

Support for Huge Pages: CryptoMMU can efficiently support huge pages, providing a performance boost for high-memory workloads and reduced verification traffic due to the large spatial coverage. For example, using a 1GB page instead of a 4KB page can have higher spatial coverage up to 262,144×. In contrast, Border Control keeps the protection table granularity at 4KB, requiring updates to 262,144 protection table entries. It has lesser spatial coverage compared to CryptoMMU due to a smaller caching granularity. Irregular access patterns within a huge page eventually result in more frequent evictions from the BCC. Therefore, design choices like larger protection table granularity would limit flexibility.

8 RELATED WORK

Secure Translation in Accelerator-rich Architectures: Modern processors are equipped with IOMMUs, facilitating efficient address translation and data isolation among devices. However, existing solutions, such as [50], [59], and [4] primarily, focus on improving IOMMU capabilities for a limited number of devices (e.g., Ethernet controllers or PCIe SSD controllers); these approaches do not address the case of larger domain-specific accelerator pools. In contrast, CryptoMMU can support an extensive array of external devices and domain-specific accelerators.

Hardware-Software Co-design of IOMMU: To mitigate the bottleneck incurred by IOMMU, a hardware and software co-design methodology has been proposed. In [73], a software-centric framework is proposed to support shared virtual memory for FPGA-included in heterogeneous systems. In this study, a configurable IOMMU, which Linux kernel API manages, can walk the host page table that can be deployed on FPGA for better performance. Moreover, a customized TLB is proposed for applications deployed on FPGA fabric, where the latency or capacity can be tuned. However, this framework has no access control mechanism, leading to an extremely vulnerable design.

Industry Patents and Patent Applications Related to Address Authentication: Leveraging address authentication and cryptographic means to realize access control appears at a high level in recent industrial patent applications and patents [35, 40]. CryptoMMU builds upon similar primitives; however, CryptoMMU is the first academic work that thoroughly evaluates the design space of cryptographic access control, demystifies its impact on hardware and software, and uniquely explores how to realize it in legacy accelerators. Finally, CryptoMMU uniquely addresses the serialization of access control checking and memory access latency through speculative read requests.

9 CONCLUSION

Growing demands for integrating third-party IPs pose new threats not only in the data center but also in edge devices. Therefore, the role of IOMMU becomes crucial to achieve a scalable and high-performance system, as it is responsible for both address translation and security preservation. In this study, CryptoMMU, an innovative cryptography- and hardware-based IOMMU, is proposed to facilitate secure and scalable integration of untrusted accelerators. It supports a wide range of architecture, system, and programming models while ensuring maximum security with negligible performance, area, and storage penalties.

Our evaluation demonstrates that CryptoMMU yields 1.13× speedup compared to the state-of-the-art scheme (i.e., Border Control). It incurs only a 1.73% slowdown compared to the unsecure ATS-only IOMMU. CryptoMMU explores a large design space, including performance, area, storage, and ease of integration. CryptoMMU will promote the adoption of access control mechanisms in SoC integration.

ACKNOWLEDGMENTS

We appreciate the anonymous reviewers for their valuable comments to improve the quality of our paper. Part of this work was funded through Office of Naval Research (ONR) grants N00014-21-1-2809 and N00014-21-1-2811, and the National Science Foundation (NSF) grants CNS-1814417, CNS-1908471, and CNS-2008339. The views, opinions, and/or findings expressed are those of the authors and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Approved for public release. Distribution is unlimited.

REFERENCES

[1] Shaizeen Aga and Satish Narayanasamy. 2017. InvisiMem: Smart memory defenses for memory bus side channel. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 94–106. https://doi.org/10.1145/3079856.3080232

[2] Nadav Amit. 2017. Optimizing the TLB Shootdown Algorithm with Page Access Tracking. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, USA, 27–39.

[3] Nadav Amit, Muli Ben-Yehuda, IBM Research, Dan Tsafrir, and Assaf Schuster. 2011. vIOMMU: Efficient IOMMU Emulation. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. USENIX Association, Portland, OR.

[4] Nadav Amit, Muli Ben-Yehuda, and Ben-Ami Yassour. 2010. IOMMU: Strategies for Mitigating the IOTLB Bottleneck. (06 2010).

[5] Roberto Avanzi. [n. d.]. The QARMA Block Cipher Family. https://eprint.iacr.org/2016/444.pdf

[6] Amro Awad, Arkaprava Basu, Sergey Blagodurov, Yan Solihin, and Gabriel H. Loh. 2017. Avoiding TLB Shootdowns Through Self-Invalidating TLB Entries. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 273–287. https://doi.org/10.1109/PACT.2017.38

[7] Amro Awad, Yipeng Wang, Deborah Shands, and Yan Solihin. 2017. ObfusMem: A low-overhead access obfuscation for trusted memories. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 107–119. https://doi.org/10.1145/3079856.3080230

[8] Stefan Balogh and Miroslav Mydlo. 2013. New possibilities for memory acquisition by enabling DMA using network card. In *2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS)*, Vol. 02. 635–639. https://doi.org/10.1109/IDAACS.2013.6663002

[9] Andrew Bean, Nachiket Kapre, and Peter Cheung. 2015. G-DMA: improving memory access performance for hardware accelerated sparse graph computation. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. 1–6. https://doi.org/10.1109/ReConFig.2015.7393317

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[11] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A quantitative analysis on microarchitectures of modern CPU-FPGA platforms. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2897937.2897972

[12] Jason Cong, Zhenman Fang, Michael Gill, and Glenn Reinman. 2015. PARADE: A cycle-accurate full-system simulation Platform for Accelerator-Rich Architectural Design and Exploration. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 380–387. https://doi.org/10.1109/ICCAD.2015.7372595

[13] Jason Cong, Zhenman Fang, Yuchen Hao, and Glenn Reinman. 2017. Supporting Address Translation for Accelerator-Centric Architectures. In . https://doi.org/10.1109/HPCA.2017.19

[14] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. 2014. Accelerator-rich architectures: Opportunities and progresses. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2593069.2596667

[15] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30, 4 (2011), 473–491. https://doi.org/10.1109/TCAD.2011.2110592

[16] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual: Volume 3A: System Programming Guide, Part 1*. Intel Corporation, Santa Clara, CA. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html

[17] Intel Corporation. 2021. *Remote Action Request*. Intel Corporation, Santa Clara, CA. https://www.intel.com/content/dam/develop/external/us/en/documents/341431-remote-action-request-white-paper.pdf

[18] Emilio G. Cota, Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. 2015. An analysis of accelerator coupling in heterogeneous architectures. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2744794

[19] John Viega David A. McGrew. [n. d.]. The Galois/Counter Mode of Operation (GCM). https://csrc.nist.rip/groups/ST/toolkit/BCM/documents/proposedmodes/gcm/gcm-spec.pdf

[20] Shijin Duan, Wenhao Wang, Yukui Luo, and Xiaolin Xu. 2021. A Survey of Recent Attacks and Mitigation on FPGA Systems. In *2021 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 284–289. https://doi.org/10.1109/ISVLSI51109.2021.00059

[21] Maik Ender, Amir Moradi, and Christof Paar. 2020. The Unpatchable Silicon: A Full Break of the Bitstream Encryption of Xilinx 7-Series FPGAs. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1803–1819. https://www.usenix.org/conference/usenixsecurity20/presentation/ender

[22] Farzad Fatollahi-Fard, David Donofrio, John Shalf, John Leidel, Xi Wang, and Yong Chen. 2017. OpenSoC system architect: An open toolkit for building soft-cores on FPGAs. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–1. https://doi.org/10.23919/FPL.2017.8056791

[23] Jayneel Gandhi, Mark D. Hill, and Michael M. Swift. 2016. Agile Paging: Exceeding the Best of Nested and Shadow Paging. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 707–718. https://doi.org/10.1109/ISCA.2016.67

[24] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 769–774. https://doi.org/10.1109/DAC18074.2021.9586216

[25] Shay Gueron. 2016. Memory Encryption for General-Purpose Processors. *IEEE Security & Privacy* 14, 6 (2016), 54–62. https://doi.org/10.1109/MSP.2016.124

[26] Krishnendu Guha, Debasri Saha, and Amlan Chakrabarti. 2020. Blockchain Technology Enabled Pay Per Use Licensing Approach for Hardware IPs. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1618–1621. https://doi.org/10.23919/DATE48585.2020.9116526

[27] SreeCharan Gundabolu and Xiaofang Wang. 2018. On-chip Data Security Against Untrustworthy Software and Hardware IPs in Embedded Systems. In *2018 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 644–649. https://doi.org/10.1109/ISVLSI.2018.00122

[28] Kosuke Hamaguchi, Shu Takemoto, Yusuke Nozaki, and Masaya Yoshikawa. 2022. Tweakable Cryptography-Based Physically Unclonable Function. In *2022 7th International Conference on Information and Network Technologies (ICINT)*. 42–45. https://doi.org/10.1109/ICINT55083.2022.00014

[29] Toshihiro Hanawa, Yuetsu Kodama, Taisuke Boku, and Mitsuhisa Sato. 2013. Tightly Coupled Accelerators Architecture for Minimizing Communication Latency among Accelerators. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. 1030–1039. https://doi.org/10.1109/IPDPSW.2013.226

[30] Festus Hategekimana, Joel Mandebi Mbongue, Md Jubaer Hossain Pantho, and Christophe Bobda. 2018. Secure Hardware Kernels Execution in CPU+FPGA

Heterogeneous Cloud. In *2018 International Conference on Field-Programmable Technology (FPT)*. 182–189. https://doi.org/10.1109/FPT.2018.00035

[31] John L. Hennessy. 2018. A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development. *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)* (2018), 27–29.

[32] Bongjoon Hyun, Youngeun Kwon, Yujeong Choi, John Kim, and Minsoo Rhu. 2020. NeuMMU: Architectural Support for Efficient Address Translations in Neural Processing Units. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1109–1124. https://doi.org/10.1145/3373376.3378494

[33] Herman Isa and Muhammad Reza Z'aba. 2014. Randomness of the PRINCE block cipher. In *International Conference on Frontiers of Communications, Networks and Applications (ICFCNA 2014 - Malaysia)*. 1–6. https://doi.org/10.1049/cp.2014.1414

[34] Nisha Jacob, Carsten Rolfes, Andreas Zankl, Johann Heyszl, and Georg Sigl. 2017. Compromising FPGA SoCs using malicious hardware blocks. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. 1122–1127. https://doi.org/10.23919/DATE.2017.7927157

[35] Kaushal Agarwal Jonathon EVANS. 2023. Preventing unauthorized translated access using address signing.

[36] Sudeendra Kumar K., Sauvagya Sahoo, Abhishek Mahapatra, Ayas Kanta Swain, and K.K. Mahapatra. 2017. A Flexible Pay-per-Device Licensing Scheme for FPGA IP Cores. In *2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. 677–682. https://doi.org/10.1109/ISVLSI.2017.123

[37] Nachiket Kapre, Han Jianglei, Andrew Bean, Pradeep Moorthy, and Siddhartha. 2015. GraphMMU: Memory Management Unit for Sparse Graph Accelerators. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 113–120. https://doi.org/10.1109/IPDPSW.2015.101

[38] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the walkers accelerating index traversals for in-memory databases. In *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 468–479.

[39] Sandhya Koteshwara, Amitabh Das, and Keshab K. Parhi. 2017. Performance comparison of AES-GCM-SIV and AES-GCM algorithms for authenticated encryption on FPGA platforms. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*. 1331–1336. https://doi.org/10.1109/ACSSC.2017.8335570

[40] Michael Kounavis, David Koufaty, Anna Trikalinou, and Rupin Vakharwala. 2021. Secure address translation services using message authentication codes and invalidation tracking.

[41] Maysam Lavasani, Hari Angepat, and Derek Chiou. 2014. An FPGA-based In-Line Accelerator for Memcached. *IEEE Computer Architecture Letters* 13, 2 (2014), 57–60. https://doi.org/10.1109/L-CA.2013.17

[42] Wang Lie and Wu Feng-yan. 2009. Dynamic Partial Reconfiguration in FPGAs. In *2009 Third International Symposium on Intelligent Information Technology Application*, Vol. 2. 445–448. https://doi.org/10.1109/IITA.2009.334

[43] Kevin Lim, David Meisner, Ali G. Saidi, Parthasarathy Ranganathan, and Thomas F. Wenisch. 2013. Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA '13)*. New York, NY, USA, 36–47. https://doi.org/10.1145/2485922.2485926

[44] Ya Liu, Tiande Zang, Dawu Gu, Fengyu Zhao, Wei Li, and Zhiqiang Liu. 2020. Improved Cryptanalysis of Reduced-Version QARMA-64/128. *IEEE Access* 8 (2020), 8361–8370. https://doi.org/10.1109/ACCESS.2020.2964259

[45] Bo Luo, Yu Li, Lingxiao Wei, and Qiang Xu. 2019. On Functional Test Generation for Deep Neural Network IPs. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 1010–1015. https://doi.org/10.23919/DATE.2019.8715042

[46] Yukui Luo, Cheng Gongye, Yunsi Fei, and Xiaolin Xu. 2021. DeepStrike: Remotely-Guided Fault Injection Attacks on DNN Accelerator in Cloud-FPGA. 295–300. https://doi.org/10.1109/DAC18074.2021.9586262

[47] Sheng Ma, Libo Huang, Yuanwu Lei, Yang Guo, and Zhiying Wang. 2019. An Efficient Direct Memory Access (DMA) Controller for Scientific Computing Accelerators. In *2019 IEEE International Symposium on Circuits and Systems (ISCAS)*. 1–5. https://doi.org/10.1109/ISCAS.2019.8702172

[48] Sheng Ma, Yuanwu Lei, Libo Huang, and Zhiying Wang. 2019. MT-DMA: A DMA Controller Supporting Efficient Matrix Transposition for Digital Signal Processing. *IEEE Access* 7 (2019), 5808–5818. https://doi.org/10.1109/ACCESS.2018.2889558

[49] Steffen Maass, Mohan Kumar, Taesoo Kim, Tushar Krishna, and Abhishek Bhattacharjee. 2020. ECO TLB: Eventually Consistent TLBs. *ACM Transactions on Architecture and Code Optimization* 17 (11 2020), 1–24. https://doi.org/10.1145/3409454

[50] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. 2015. RIOMMU: Efficient IOMMU for I/O Devices That Employ Ring Buffers. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 355–368. https://doi.org/10.1145/2694344.2694355

[51] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. 2020. Agile SoC Development with Open ESP : Invited Paper. In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9.

[52] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023)*. New York, NY, USA, 742–755. https://doi.org/10.1145/3582016.3582063

[53] Maria I. Mera Collantes and Siddharth Garg. 2020. Do Not Trust, Verify: A Verifiable Hardware Accelerator for Matrix Multiplication. *IEEE Embedded Systems Letters* 12, 3 (2020), 70–73. https://doi.org/10.1109/LES.2019.2953485

[54] Benoît Morgan, Eric Alata, Vincent Nicomette, and Mohamed Kaaniche. 2018. IOMMU protection against I/O attacks: A vulnerability and a proof-of-concept. *Journal of the Brazilian Computer Society* 24 (12 2018). https://doi.org/10.1186/s13173-017-0066-7

[55] Pascal Nasahl, Robert Schilling, and Stefan Mangard. 2021. Protecting Indirect Branches Against Fault Attacks Using ARM Pointer Authentication. In *2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. 68–79. https://doi.org/10.1109/HOST49136.2021.9702268

[56] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W. Moore. 2018. Understanding PCIe Performance for End Host Networking. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 327–341. https://doi.org/10.1145/3230543.3230560

[57] Lena E. Olson, Jason Power, Mark D. Hill, and David A. Wood. 2015. Border control: Sandboxing accelerators. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 470–481. https://doi.org/10.1145/2830772.2830819

[58] Lena E. Olson, Simha Sethumadhavan, and Mark D. Hill. 2016. Security Implications of Third-Party Accelerators. *IEEE Computer Architecture Letters* 15, 1 (2016), 50–53. https://doi.org/10.1109/LCA.2015.2445337

[59] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. 2015. Utilizing the IOMMU Scalably. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 549–562.

[60] Bharath Pichai, Lisa Hsu, and Abhishek Bhattacharjee. 2014. Architectural Support for Address Translation on GPUs: Designing Memory Management Units for CPU/GPUs with Unified Address Spaces. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, 743–758. https://doi.org/10.1145/2541940.2541942

[61] Simon Pickartz, Pablo Reble, Carsten Clauss, and Stefan Lankes. 2014. SWIFT: A Transparent and Flexible Communication Layer for PCIe-Coupled Accelerators and (Co-)Processors. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops*. 371–380. https://doi.org/10.1109/IPDPSW.2014.48

[62] Jason Power, Mark D. Hill, and David A. Wood. 2014. Supporting x86-64 address translation for 100s of GPU lanes. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 568–578. https://doi.org/10.1109/HPCA.2014.6835965

[63] Nitin Pundir, Fahim Rahman, Farimah Farahmandi, and Mark Mohammad Tehranipoor. 2021. What is All the FaaS About? - Remote Exploitation of FPGA-as-a-Service Platforms. *IACR Cryptol. ePrint Arch.* 2021 (2021), 746.

[64] Qualcomm Inc. [n. d.]. *Pointer Authentication on ARMv8.3*. Technical Report.

[65] Jeyavijayan Rajendran, Vivekananda Vedula, and Ramesh Karri. 2015. Detecting malicious modifications of data in third-party intellectual property cores. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1145/2744769.2744823

[66] Anup Rani, Arun Pai, Bhushan Naware, Zong Han Yang, and Tsue-Yi Huang. 2020. Direct Memory Access Remapping for Thunderbolt, Feature Deployment at Platform Level. In *2020 IEEE International Conference for Innovation in Technology (INOCON)*. 1–5. https://doi.org/10.1109/INOCON50539.2020.9298289

[67] Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, Giorgiomaria Cicero, and Giorgio Buttazzo. 2020. AXI HyperConnect: A Predictable, Hypervisor-level Interconnect for Hardware Accelerators in FPGA SoC. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC18072.2020.9218652

[68] Mohamad Alfadl Rihani, Fabienne Nouvel, Jean-Christophe Prévotet, Mohamad Mroue, Jordane Lorandel, and Yasser Mohanna. 2016. Dynamic and partial reconfiguration power consumption runtime measurements analysis for ZYNQ SoC devices. In *2016 International Symposium on Wireless Communication Systems (ISWCS)*. 592–596. https://doi.org/10.1109/ISWCS.2016.7600973

[69] Paul D. Rosero-Montalvo. 2021. A Survey on Security Concerns and Their Actual Solutions for using FPGAs in Cloud Computing. In *2021 IEEE Latin American Conference on Computational Intelligence (LA-CCI)*. 1–6. https://doi.org/10.1109/LA-CCI48322.2021.9769794

[70] Fernand Lone Sang, Vincent Nicomette, and Yves Deswarte. 2011. I/O Attacks in Intel PC-based Architectures and Countermeasures. In *2011 First SysSec Workshop*. 19–26. https://doi.org/10.1109/SysSec.2011.10

[71] Mitali Sinha, Pramit Bhattacharyya, Sidhartha Sankar Rout, Neha Bhairavi Prakriya, and Sujay Deb. 2022. Securing an Accelerator-Rich System From Flooding-Based Denial-of-Service Attacks. *IEEE Transactions on Emerging Topics in Computing* 10, 2 (2022), 855–869. https://doi.org/10.1109/TETC.2021.3049826

[72] Carlos Villavieja, Vasileios Karakostas, Lluis Vilanova, Yoav Etsion, Alex Ramirez, Avi Mendelson, Nacho Navarro, Adrian Cristal, and Osman S. Unsal. 2011. DiDi: Mitigating the Performance Impact of TLB Shootdowns Using a Shared TLB Directory. In *2011 International Conference on Parallel Architectures and Compilation Techniques*. 340–349. https://doi.org/10.1109/PACT.2011.65

[73] Pirmin Vogel, Andrea Marongiu, and Luca Benini. 2019. Exploring Shared Virtual Memory for FPGA Accelerators with a Configurable IOMMU. *IEEE Trans. Comput.* 68, 4 (2019), 510–525. https://doi.org/10.1109/TC.2018.2879080

[74] Riad S. Wahby, Max Howald, Siddharth Garg, Abhi Shelat, and Michael Walfish. 2016. Verifiable ASICs. In *2016 IEEE Symposium on Security and Privacy (SP)*. 759–778. https://doi.org/10.1109/SP.2016.51

[75] Jooho Wang, Sungkyung Park, and Chester Sungchung Park. 2021. Optimization of Communication Schemes for DMA-Controlled Accelerators. *IEEE Access* 9 (2021), 139228–139247. https://doi.org/10.1109/ACCESS.2021.3118094

[76] Susila Windarta, Kalamullah Ramli, and Dodi Sudiana. 2020. Security Evaluation of LIGHTMAC: Second Preimage Attack using Existential Forgery. In *2020 1st International Conference on Information Technology, Advanced Mechanical and Electrical Engineering (ICITAMEE)*. 265–269. https://doi.org/10.1109/ICITAMEE50454.2020.9398503

[77] Jin Xu, Dayin Wang, Dongdai Lin, and Wenling Wu. 2006. An Efficient One-key Carter-Wegman Message Authentication Code. In *2006 International Conference on Computational Intelligence and Security*, Vol. 2. 1331–1334. https://doi.org/10.1109/ICCIAS.2006.295275

[78] Mao Ye, Clayton Hughes, and Amro Awad. 2018. Osiris: A Low-Cost Mechanism to Enable Restoration of Secure Non-Volatile Memories. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 403–415. https://doi.org/10.1109/MICRO.2018.00040

[79] Efe Berkay Yitim and Ece Güran Schmidt. 2022. An AXI Data Shaper for Heterogeneous FPGA System-on-Chip (SoC) Architectures. In *2022 30th Signal Processing and Communications Applications Conference (SIU)*. 1–4. https://doi.org/10.1109/SIU55565.2022.9864667

[80] Shaza Zeitouni, Ghada Dessouky, and Ahmad-Reza Sadeghi. 2020. SoK: On the Security Challenges and Risks of Multi-Tenant FPGAs in the Cloud. arXiv:cs.CR/2009.13914

[81] Yanqi Zhou, Xuanyi Dong, Tianjian Meng, Mingxing Tan, Berkin Akin, Daiyi Peng, Amir Yazdanbakhsh, Da Huang, Ravi Narayanaswami, and James Laudon. 2022. Towards the Co-design of Neural Networks and Accelerators. In *Proceedings of Machine Learning and Systems*, D. Marculescu, Y. Chi, and C. Wu (Eds.), Vol. 4. 141–152.

[82] Kazi Abu Zubair and Amro Awad. 2019. Anubis: Ultra-Low Overhead and Recovery Time for Secure Non-Volatile Memories. In *Proceedings of the 46th International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/3307650.3322252

A ARTIFACT APPENDIX

A.1 Abstract

This artifact provides a virtual machine environment that contains different ready-to-build-and-run IOMMU models for the evaluation results provided in Section 6.1 and Section 6.2 (i.e., Figure 9 - Figure 11). The models are implemented by extending the gem5-parade simulator as described in Section 5. In this section, we provide detailed instructions to set up the environment and run each model.

A.2 Artifact check-list (meta-information)

Our artifact is provided with a virtual machine (VM) image; hence, the checklist below indicates requirements on the guest OS. Please note that all the necessary tools are already installed in the VM image, which is made on VMWare Workstation 14.

- **Program:** Five directories are provided: Full IOMMU, ATS-only IOMMU, Border Control, CryptoMMU, and CryptoMMU(Read Acc.)
- **Compilation:** GCC/G++ 4.8.5, Python 2.7.17, Scons-1.3.1, Swig-2.0.9, Protobuf-2.5.0, Mono-6.12.0.200
- **Run-time environment:** Ubuntu 18.04.06 LTS Desktop 64-bit
- **Hardware:** The machine whose main memory is larger than 32 GB (64 GB is recommended for a VM)
- **Metrics:** Total execution time
- **Output:** Simulation statistics
- **Experiments:** Provided scripts
- **How much disk space required (approximately)?:** 38 GB for a virtual machine image
- **How much time is needed to complete experiments (approximately)?:** around 1-4 hours per experiment (Full IOMMU takes much longer)
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD-3
- **Workflow framework used?:** No
- **Other utilities installed in VM (after doing `apt update` and `apt upgrade`):** build-essential, vim, git, m4, zlib1g, zlib1g-dev, libgoogle-perftools-dev, python-dev, python, libtool, kmod, uuid-dev, libfabric-dev, cmake, glib2.0, bison, doxygen, perl, libtool-ltdl, valgrind, htop, iptables, libhdf5, patch

A.3 Description

We provide virtual machine images of our CryptoMMU infrastructure used in this paper on Zenodo.

A.3.1 How to access. Both VMX (for VMWare) and OVF-based (for VirtualBox) VM images can be downloaded from the Zenodo link: https://doi.org/10.5281/zenodo.8287142

CryptoMMU is evaluated under the full system simulation mode, which requires disk image and kernel binaries that contain precompiled workloads. All these requirements are also included in the provided VM images. We used the disk image and kernel binaries available on the original gem5-parade repo:
http://www.sfu.ca/~zhenman/files/software/disk-binary.tar.gz

A.3.2 Hardware dependencies. There is no restriction on CPUs, as we only need an environment that can run VMWare. However, it is recommended to have main memory larger than 32 GB for the VM, because each workload consumes at least 16 GB of main memory, moreover, some workloads of Full IOMMU (e.g., Denoise) consume nearly 32 GB of main memory.

A.3.3 Software dependencies. For artifact evaluation, the simulation infrastructure is sensitive to not only dependencies but also operating systems, because some certain versions of dependencies are not supported on newer operating systems. Therefore, we made a VM image wherein the guest OS is Ubuntu 18.04.06 LTS Desktop 64-bit, which is similar to our simulation infrastructure built on the real machine.

Our application has several dependencies of legacy packages. The packages that need to be installed to build and run the simulation are listed as follows:

- GCC/G++-4.8.5
- Python 2.7.17
- Scons-1.3.1
- Swig-2.0.9
- Protobuf-2.5.0
- Mono-6.12.0.200

Regarding the compilation of benchmarks, the older version of GCC is required (i.e., GCC/G++-4.1.2), since gem5-parade simulates an older version of Linux OS.

A.3.4 Workloads. The workloads that we simulated in our experiment are present in the disk image. It is not required to re-compile the workloads. There is a shell script that is provided in each repository to run these workloads.

A.4 Installation

Here are the steps to prepare the VM on VMWare.

(1) Extract `Ubuntu18_64_vmx.tar.gz` and get a directory `Ubuntu18_64/`
(2) Open VMWare software. Under the GUI of VMWare Workstation, click **File-Open** and load the VM image by navigating to `/path/to/Ubuntu18_64/Ubuntu18_64.vmx`. Also, please provision the main memory size and the number of cores to the VM by navigating to **VM-Settings-Memory**
(3) Start up the VM by clicking **VM-Power-Start Up Guest** under the GUI of VMWare Workstation. You will see a pop-up message and select "*I Copied It*" option, which generates a new UUID and MAC address; this procedure ensures no conflicts in the network
(4) Login to VM, wherein the ID and the password are *saca* and *cryptommu*, respectively

Here are the steps to prepare the VM on VirtualBox.

(1) Extract `Ubuntu18_64_ovf.tar.gz` and get a directory `Ubuntu18_64/`
(2) Under the GUI of VirtualBox, click **File-Import Appliance** and load the VM image by navigating to `/path/to/Ubuntu18_64/Ubuntu18_64.ovf`. Under the CLI, you can import the image by typing the command below (e.g., dual-core with 16 GB main memory):
```
$ VBoxManage import /path/to/Ubuntu18_64_ovf/Ubuntu18_64.ovf
--vsys 0 --vmname vm_cryptommu --cpus 2 --memory
16384 --basefolder /path/to/generate/your/vm
```
(3) Sometimes, you need to explicitly specify the graphics controller for the VM if the default controller (i.e., VBoxVGA) is not supported on some host OSes by navigating to **Machine-Settings-Displays-Graphics Controller-VMSVGA**, or:

```
$ VBoxManage modifyvm "vm_cryptommu" --graphicscon
troller vmsvga
```

(4) Start up the VM by clicking **Machine-Start-Normal Start** under the GUI of VirtualBox. You can also access the VM in headless mode by leveraging tools, such as *rdesktop* or *ssh*, after opening the session with the following command line:
```
$ VBoxManage startvm "vm_cryptommu" --type headless
```
(5) Login to VM, wherein the ID and the password are *saca* and *cryptommu*, respectively

Here are the steps to build simulation infrastructure **on the VM**:

(1) Open the terminal and directly head to the objective directory for evaluation. Say CryptoMMU:
```
$ cd CrytoMMU-sim-artifact/CryptoMMU
```
(2) Set environment variables as below:
```
$ export PARADE_HOME=$(pwd)
$ export M5_PATH=$PARADE_HOME
```
(3) Build the binary by running the provided script (Please remove the existing `build/` directory if you want to build from the blank-whiteboard status):
```
$ ./build.gem5.sh
```

Making checkpoints: The VM image contains the checkpoint of the simulator, which contains the simulation snapshot after the initial startup and booting of Linux OS. The checkpoint is present in the directory, `ckpt-1core/`. Additionally, we have provided steps to generate the checkpoint (*not required* in our VM image).

(1) Download the master repository of gem5-parade
```
$ git clone https://github.com/cdsc-github/parade-ara-simulator.git
```
(2) Enable `SIM_DEDICATED_ARA` macro in three files: `src/mem/ruby/profiler/Profiler.cc`, `src/mem/ruby/system/System.cc`, and `src/sim/simulate.cc`
(3) Build the binary as follow:
```
$ ./build.gem5.sh
```
(4) Run the binary to generate the checkpoint:
```
$ build/X86/gem5.opt configs/example/fs.py --cpu-type=atomic -n 1 --mem-size=2GB --script=configs/boot/hack_back_ckpt.rcS
```
(5) Copy the generated checkpoint to the objective directory looked up by gem5. If your working directory is in `CrytoMMU-sim-artifact/CryptoMMU/`:
```
$ cp -r cpt.5176168078500 ckpt-1core/
```

A.5 Experiment workflow

`run_bench.sh` is used to simulate all five models. The output files will be generated at the directory, `result_cXaY/TDLCA_Z`, where X, Y, and Z denote the cache size, the number of accelerators, and the workload name, respectively; `-h` option can be used to infer these X, Y, and executable workloads. For example, we can run a workload, *Denoise*, by typing the command below (Please note that X=8 and Y=64 as default values if no explicit option is conferred):
```
$ ./run_bench.sh Denoise -c X -a Y
```

Below items are the explanation of important macros in `run_bench.sh`:

(1) `BENCH_LIST`: It contains all benchmarks that have been run in our evaluation. Please run with "-h" option to observe runnable workloads in this artifact
(2) `DIR_OUT_BENCH`: It contains all output files, including stats (i.e., `stats.txt`), for each benchmark. The output messages will be dumped as `result.txt` in each benchmark directory. In this artifact, this directory is formatted as `result_cXaY/TDLCA_Z`
(3) `FILE_BOOTSCRIPT`: It contains the boot scripts of full system simulation for each benchmark (i.e., .rCS file)
(4) `DIR_CKPT`: It contains the simulation checkpoint generated in the previous section
(5) `CFG_OPTIONS`: It indicates the common gem5-parade configuration parameters in our evaluation

Running different models: After running the CryptoMMU model, you may want to run the binary of a different model. To do this, you need to update environment variables (i.e., `PARADE_HOME` and `M5_PATH`). Say *Border Control*:
```
$ export PARADE_HOME=/path/to/Border_Control
$ export M5_PATH=$PARADE_HOME
$ /path/to/Border_Control/run_bench.sh Denoise
```

A.6 Experiment customization

To evaluate different scenarios presented in Section 6.2, we provide two optional configuration options to propagate values to the following flags of gem5-parade in `run_bench.sh`:

(1) "-c [*val*]": The LLC size per bank (in KB), which will be propagated to the flag `--l2_size`. In our environment, the number of LLC banks is fixed at 32; hence, a 2 MB of LLC should be configured as `-c 64`
(2) "-a [*val*]": The number of accelerators, which will be propagated to the flag `--num_accinstance`

A.7 Evaluation and expected results

Our artifact is provided to reproduce Figure 9 - Figure 11 in Section 6.1 and Section 6.2. In these figures, we compared the *total execution time* of the benchmark, which is the value associated with `system.switch_cpus.numCycles` in `stats.txt`; this value is the total CPU cycles simulated after restoring the checkpoint.

After running the evaluation for all configurations, the normalized performance values can be obtained by taking the inverse of Border Control's total execution time as a denominator and the inverse of other models' total execution time as numerators (Equivalently, the Border Control's execution time as numerator and the others' execution time as denominator). For Figure 9, *Denoise* will show the normalized values of 0.10, 1.13, 1.23, and 1.27 with respect to Border Control for Full IOMMU, CryptoMMU, CryptoMMU(Read Acc.), and ATS-only IOMMU, respectively; these values are close to the bars presented in Figure 9.

Similarly, for Figure 10 and Figure 11, the execution time values should be normalized with the execution time values of Border Control, which are configured as the corresponding number of accelerators and LLC size, respectively. For example, in terms of the bar AFU 4 in Figure 11, it is equivalent to the total execution time of *Border Control* divided by the total execution time of *CryptoMMU (Read Acc.)*; both are configured as *four* accelerators.